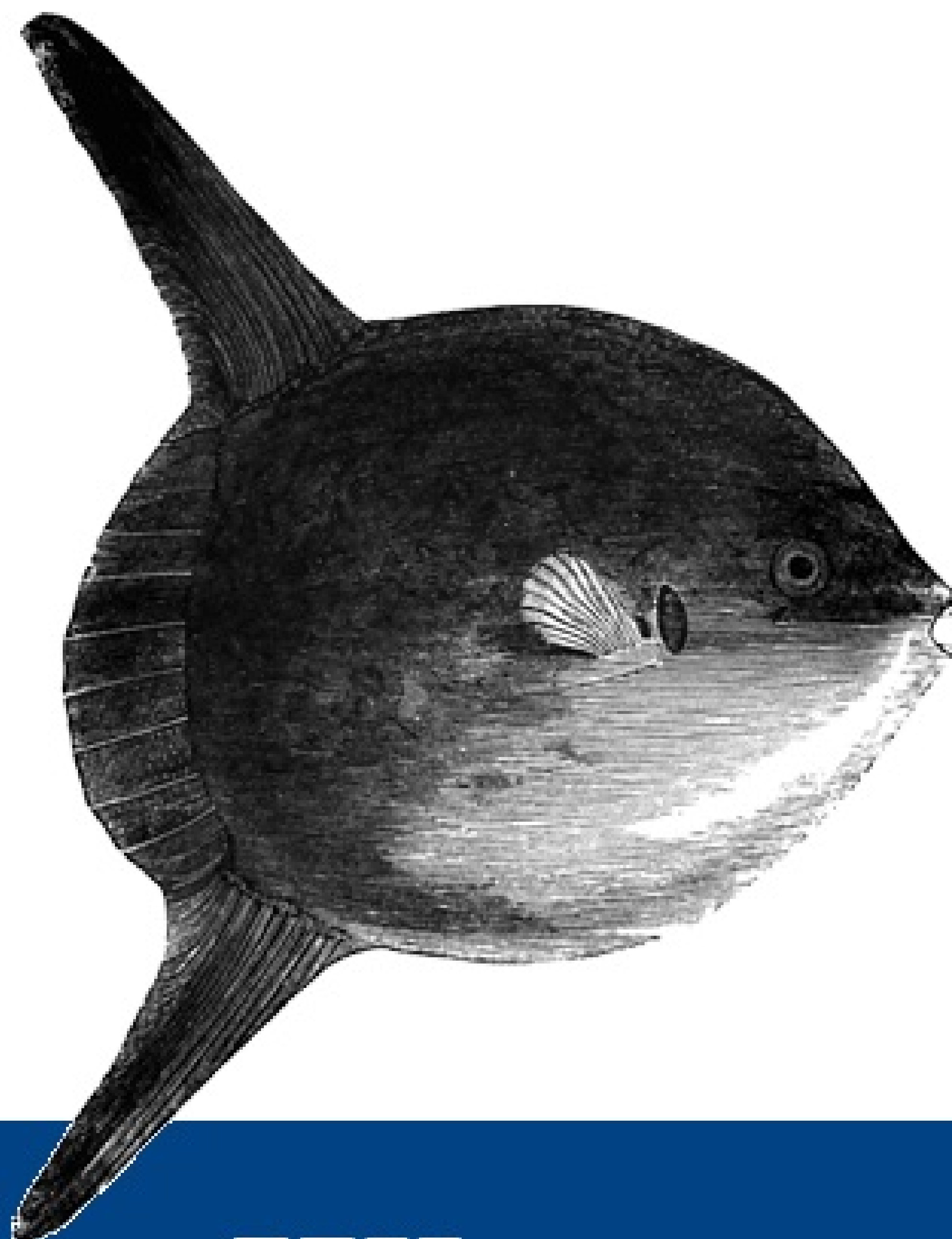
Another Flask Book For Beginner



The Way To Flask

The Best Practice

O RLY?

Liqiang Lau

目錄

The Way To Flask

本文目标

通过讲解 Flask 以及它的扩展们，介绍通用用法以及使用过程中的问题和坑，帮助读者使用 Python 编程语言快速得开发健壮的 Web（API）服务端程序。本书在编写之初以及编写过程中始终坚持以下几条原则：

- 让 Python 初学者/会其他语言但没用过 Python 的人能快速入手
- 循序渐进得让读者感受 Flask 的简便与强大
- 以生动有趣的语言讲述 Flask 从入门到着迷

Flask 简介

Flask 是一个使用 Python 编写的轻量级 Web 应用框架，核心的思想就是自身尽可能提供少的东西，作为一个微框架，将更多的内容以插件的形式提供，因此，衍生出了一系列以 Flask 为核心的 插件。截止至2016年06月02日，在 Github 上已有 **20730** 个星，**6426** 个 Fork 以及 **1511** 个 Watch。

通过使用 pip 包管理工具统计，Flask 的扩展已经达到 **800+**，涵盖大部分日常工作使用到的内容。

声明

联系我

如果对本书提到的知识点有不解或者觉得有误，可以根据以下联系方式与我联系，同时，欢迎大家一起编撰修改本书，让更多的人能够喜爱 Flask。

- 主页：https://liuliqiang.info
- 邮箱：liqianglau@outlook.com
- Gitbook: https://luke0922.gitbooks.io/the-way-to-flask/content/
- GitHub: https://github.com/luke0922/the-way-to-flask.git

更新记录

Version 1.0

- date： 2016-6-2

- desc： 终于在一个多月的时间里完成了第一版，期间发生了很多事情，但是，还是坚持下来了，完成了第一版的《The Way To Flask》，虽然个人觉得还有很大的改进空间，但至少是有这么粗糙的一版，后面有什么问题，可以根据大家的建议进行改进。

Version 1.1

- date: 2016-6-11
- desc: 在 Pycon2016 上观看了《Flask at Scale》的讲解，对 Flask 的项目有了更多的一些理解，发现了 V1.0 的内容已经符合可维护性的要求，在这个版本中新加入优化性能的部分。

Version 1.2

- begin: 2017-03-01
- end: 2017-03-01
- desc：修改一些文档的错误

Version1.3

- begin: 2017-05-01
- end: 2017-05-01
- desc：使用 mkdocs 重构文档

第一部分

Flask 快速入门

本书概述

在 Python 中有很多优秀的 Web 开发框架，例如 Django、Flask和 Tornado等等。

每种框架都有其自身的独特之处，

- Django 自己集成了丰富的功能，将数据库模块、模板以及后台管理等模块都集成在自身内部，和框架一起打包发布；
- 而 Flask 则以最简原则，自身框架只附带很简单的路由、模板功能，而提供了简单的扩展接口，从而将其他的功能都以扩展的形式提供，从而产生了大量的强大的各种扩展，Flask也因此以扩展丰富而受欢迎；
- Tornado 则与 Django 和 Flask 走不同的道路，Tornado 的主打功能是异步请求处理，适用于 IO 操作繁多的应用。

这个系列文章的主要介绍对象就是 Flask 以及它的插件们，因此对于其他框架也就在上面简约得一言带过，有兴趣的同学可以自行查找资料学习。

本书的文章顺序主要按照以下的骨架进行介绍：

- 第一部分讲解 Flask 的基础功能
- 第二部分讲解 Flask 的几个重要插件以及注意点
- 第三部分将以前面介绍的内容综合起来实践一个 Todo 系统

为了让同学们在阅读的时候同时实践可以产生和我讲解出现一样的效果，下面我有必要罗列一下本书中使用到的数据库、Python库的版本等信息。

数据库

```
MongoDB：
    version：3.2.6
    ip：localhost
    port：27017
Redis：
    version：3.0.5
    ip：localhost
    port：6379
```

Python 依赖库

```
Flask==0.10.1
flask-mongoengine==0.7.5
Flask-Login==0.3.2
Flask-Admin==1.4.0
Flask-Redis==0.1.0
Flask-WTF==0.12
```

简单的 Flask 应用

作为本书的第一个示例，也可能是你接触的第一个 Flask 应用，我还是以程序届常规的 Hello World 为例来编写一个非常简单的例子。

这个例子的功能就是你在浏览器中输入URL：

```
http://localhost:5000
```

然后，你就可以在浏览器中看到：

```
Hello World！
```

Simple Flask App

首先，我们先来看一个简短的代码

```
#!/usr/bin/env python
# encoding: utf-8
from flask import Flask

app = Flask(__name__)


@app.route('/')
def index():
    return "Hello World!"

app.run()
```

将这段代码保存为 `app.py` ，然后再使用 python 运行这个文件：

```
python app.py
```

回车之后，你将会看到类似以下的输出：

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

如果有其它错误，你可以仔细看看是什么问题，代码和我上面是否一致，还有一个很重要的点就是，你是否已经安装了 Flask ，如果没有的话，那么安装一下：

```
pip install Flask==0.10.1
```

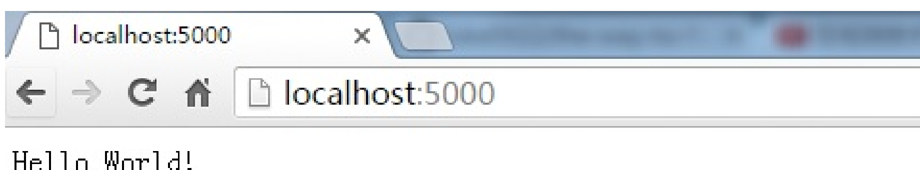
安装完成后，继续使用 Python 运行上述文件：

```
python app.py
```

然后在浏览器上打开以下URL：

```
http://localhost:5000
```

你将会看到这个界面：



Hello World!

那就说明你的第一个 Flask 应用已经运行成功了。

简析第一个应用

对于你运行成功的第一个程序很简单，我们就做一些简单的分析，让你有一个简单的了解。

前两行的编码说明就不说了，这是 Python 的特性，而不是 Flask 特有的，如果读者有不懂的话，建议查看 Python 的文件编码说明。

然后继续看代码，我们的所有代码只有一个 `import` ，就是 `Flask` ，这是 Flask 这个框架的核心，我们把它认为是服务器就可以了，目前不需要多关注：

```
from flask import Flask
```

然后接下去看，解析来一句是初始化了一个 Flask 变量，那么我们就可以认为是创建了一个服务器；需要注意的是这里传递了一个参数 **name**，我们知道在 Python 中 **name** 这个变量是表示模块的名称，这个参数对于 **Flask** 很重要，因为 **Flask** 会依赖于它去判断从哪里找模板、静态文件。

```
app = Flask(__name__)
```

接下来三句目前来说可能有点超出我们的讨论范围，但是我们这里稍微讲解一下好了，这三句中关键是第一句和第三句。

```
@app.route('/')
def index():
    return "Hello World!"
```

第一句中关键的是 `'/'` 这个参数，这个参数的作用是说下面的这个函数对应于我们在浏览器中输入的地址：

```
http://localhost:5000 + 后面的参数
```

这样说，大家可能不太明白，假设换成：

```
@app.route('/hello')
def hello():
    return "hello world"
```

的话，那么也就表示，我们在浏览器中访问：

```
http://localhost:5000/hello
```

那么 Flask 就会调用到 hello 这个函数。

那第三句的意思大家可能会比较容易理解了，没错，return 的内容就是我们在浏览器中看到的内容了。我们的代码中 return 的是 "Hello World！"，那么我们在浏览器中看到的就是 Hello World！

到目前为止，我们 import 了服务器（**import Flask**）,创建了服务器（***Flask(name)***），是时候将服务器运行起来了，是的，最后一句

```
app.run()
```

就是表示将服务器运行起来，接受浏览器的访问。

那么整个过程就是这样的，当我们在浏览器中输入

```
http://localhost:5000
```

的时候，其实浏览器默默得在我们的 URL 后面加入了一个 `/`，真实访问的就是

```
http://localhost:5000/
```

其实也就是对应着我们的

```
app.route('/')
```

函数了，这个函数

```
return "Hello World!"
```

所以我们在浏览器中就看到了：

```
Hello World！
```

简单的 REST 服务

随着移动设备的不断发展，移动端的需求日益增大，对于大多数公司来说，可能用户量已超越 PC 端。而随着移动端发展，伴随而来的是对于客户端和服务器的交互越来越轻量化，相对“笨重”的 HTML 页面逐渐被移动端抛弃（但是 H5 的出现，这一情况有所转变），而此时 REST 服务模式被越来越多人接受。

通俗来说，REST 服务最少都需要提供查询功能，丰富一下的则会提供增删改查功能，其中还可能包含批量的操作。但是，本章因为是介绍如何使用 Flask 编写一个 REST 服务器的示例，所以本章要介绍的功能是：

- 使用 PUT、DELETE、POST和GET 进行数据增删改查
- 返回 json 结构的数据

修改第一个程序

我们回忆一下第一个程序，他的功能就是我们在浏览器中输入 URL

```
http://localhost:5000
```

时，返回一个字符串“Hello World！”，于是我们就想，我们能不能将这个字符串换成 json 序列？这样不就等于我们实现了 REST 的查询 API 了？

于是，我们可能第一冲动就会这么实现：

```
#!/usr/bin/env python
# encoding: utf-8
import json
from flask import Flask

app = Flask(__name__)

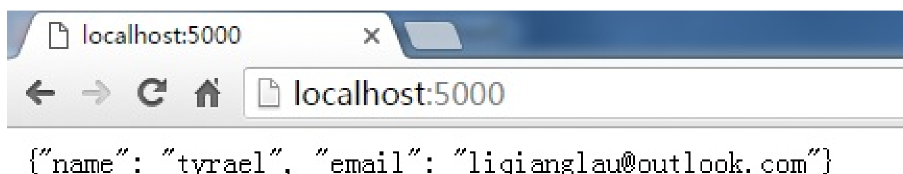

@app.route('/')
def index():
    return json.dumps({'name': 'tyrael',
                       'email': 'liqianglau@outlook.com'})

app.run()
```

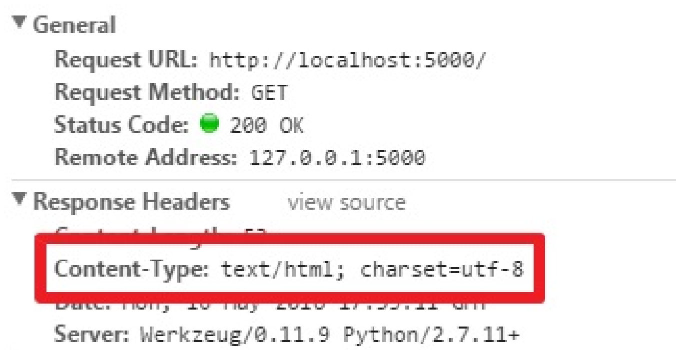
其实我们就是修改了返回的字符串，将它修改成 JSON 的字符串，然后我们在浏览器上打开

```
http://localhost:5000
```

看到的是：



哇！！好像是实现了我们想要的功能，返回了 JSON 字符串，但是我们打开 Chrome（我使用的是 Chrome，Safari 和 Firefox 同样有类似的工具）的调试工具（Windows下按： Ctrl + Alt + I，Mac 下按：Cmd + Shift + I），我们可以看到其实这个返回的数据类型居然是 html 类型：



你可能会奇怪这会有什么影响，这个影响大多数情况下应该不大，但是对于某些移动端的库，可能会根据这个响应头来处理数据，这个时候就悲剧了。

返回json

处理这个情况我们不能简单得想把这个响应头设置成 json 格式，这样修补 bug 是会导致其他 bug 的，譬如其他我们不知道的地方还有类似的坑。

更好的解决方案是使用 Flask 的 jsonify 函数，我这里使用这个函数修改一下代码：

```
#!/usr/bin/env python
# encoding: utf-8
import json
from flask import Flask, jsonify

app = Flask(__name__)

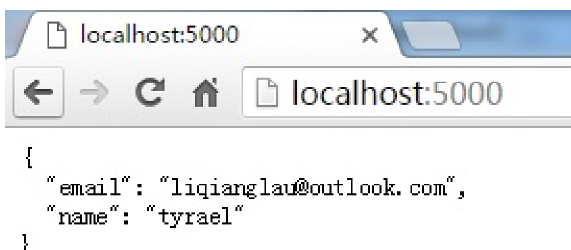

@app.route('/')
def index():
    return jsonify({'name': 'tyrael',
                    'email': 'liqianglau@outlook.com'})

app.run()
```

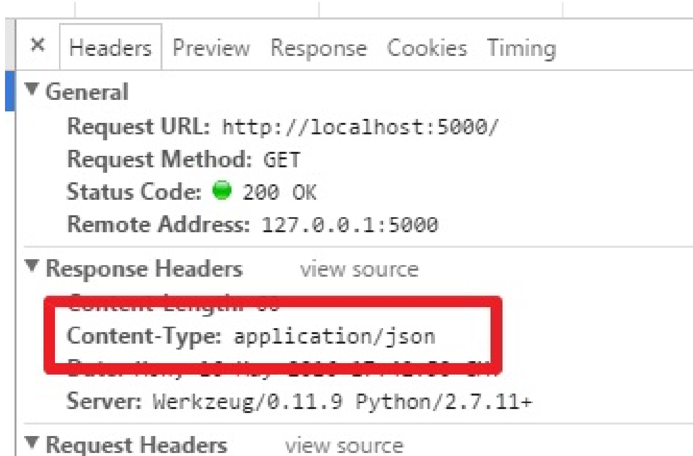
这里做了两处修改，分别是：

```
from flask import ...., jsonify
... ...
return jsonify({'name': 'tyrael',
                'email': 'liqianglau@outlook.com'})
```

此时，我们再保存代码，运行代码，并且访问看看：



我们发现代码居然排好了版式，然后再看看响应头：



响应头也变成了 `application/json` 了。

好了，那么我们这里达到了第一个目的了，返回 json 数据。但是，我们的另外一个目的—使用 DEL，PUT和 POST 方法怎么处理？

请求方法

我们知道常用的 HTTP 请求方法有 6 种，分别是

- GET
- POST
- PUT
- DELETE
- PATCH
- HEAD

那么我们刚刚的代码只能默认得处理 GET 的情况（浏览器默认使用GET），那么其他情况怎么处理？

这时我们回到我们的代码中，既然我们的 **URL** 是通过

```
app.route('...')
```

来拼接的，那么，请求方法是不是也可以在这里指定？

事实上就是这样的，请求方法通过一个叫做 methods 的参数指定，例如下面分别对应 POST、DELETE、PUT 方法。

```
@app.route('/', methods=['POST'])
@app.route('/', methods=['DELETE'])
@app.route('/', methods=['PUT'])
```

还有一个问题就是我们因为要做数据的增删改查，所以需要考虑数据的保存，因为数据库的操作在本章又是超出范围的讨论，所以这里我们简单得以文件作为保存数据的媒介。进行数据操作，那么我们的代码可以这么写：

```
#!/usr/bin/env python
# encoding: utf-8
import json
from flask import Flask, request, jsonify

app = Flask(__name__)


@app.route('/', methods=['GET'])
def query_records():
    name = request.args.get('name')
```

```
    print name
    with open('/tmp/data.txt', 'r') as f:
        data = f.read()
        records = json.loads(data)
        for record in records:
            if record['name'] == name:
                return jsonify(record)
        return jsonify({'error': 'data not found'})


@app.route('/', methods=['PUT'])
def create_record():
    record = json.loads(request.data)
    with open('/tmp/data.txt', 'r') as f:
        data = f.read()

    if not data:
        records = [record]
    else:
        records = json.loads(data)
        records.append(record)

    with open('/tmp/data.txt', 'w') as f:
        f.write(json.dumps(records, indent=2))
    return jsonify(record)


@app.route('/', methods=['POST'])
def update_record():
    record = json.loads(request.data)
    new_records = []
    with open('/tmp/data.txt', 'r') as f:
        data = f.read()
        records = json.loads(data)

    for r in records:
        if r['name'] == record['name']:
            r['email'] = record['email']
        new_records.append(r)

    with open('/tmp/data.txt', 'w') as f:
        f.write(json.dumps(new_records, indent=2))
    return jsonify(record)


@app.route('/', methods=['DELETE'])
def delte_record():
    record = json.loads(request.data)
    new_records = []
    with open('/tmp/data.txt', 'r') as f:
        data = f.read()
        records = json.loads(data)
        for r in records:
```

```
            if r['name'] == record['name']:
                continue
            new_records.append(r)

    with open('/tmp/data.txt', 'w') as f:
        f.write(json.dumps(new_records, indent=2))

    return jsonify(record)

app.run(debug=True)
```

这段代码虽然很长，但是代码都比较容易懂，而且都是比较简单的文件操作。

这段代码我们需要关注的点有以下几点：

- 如何设置请求方法

```
@app.route('/', methods=['GET'])
@app.route('/', methods=['PUT'])
@app.route('/', methods=['POST'])
@app.route('/', methods=['DELETE'])
```

- 如何获取数据

在 Flask 中有一个 request 变量，这是一个请求上下文的变量，然后里面包含多个属性是可以用来获取请求的参数的，例如我们这里用到了两种方式：

1. request.args.get('name')

   `request.args` 这个属性用于表示 GET 请求在 URL 上附带的参数

2. json.loads(request.data)

   `request.data` 这个属性用于表示 POST 等请求的请求体中的数据

我们目前对 request 变量就做这么多介绍吧，毕竟我们本章的目标是让大家了解如何处理 GET、POST、PUT 等不同的请求方式如何处理。

第二部分

Flask 插件使用指南

使用 Flask-MongoEngine 集成数据库

在前面一章 简单的 REST 服务 中，我们的数据都是保存在文件中的，我们可以发现，这样很是繁琐，每个请求中都需要进行读取文件，写出文件的操作，虽然显然我们可以对文件操作进行一个封装，但是，毕竟是文件存储，数据稍微多一点查询等操作必然时间变长。

面对这样的一个问题，这里引入了对数据库的依赖，在我们的 本书概述 中，我介绍了数据库的版本信息，本章使用的是 MongoDB，具体的版本还有数据库地址信息为：

```
version：3.2.6
ip：localhost
port：27017
```

创建数据模型

既然我们想使用数据库来保存数据，我们可以使用原生的 pymongo 来操作 MongoDB，但是，我们这里为了更进一步得简化我们的操作，所以我们需要创建数据模型。

数据模型主要的功能是用于说明我们的数据包含哪些字段，每个字段分别是什么类型，有什么属性（唯一的，还是固定几个值中的一个）等等。这样可以帮助我们在操作数据的时候可以时刻很清晰得知道我们的数据的信息，即使我们不看数据库中的数据。

这里我们要介绍的操作 MongoDB 的 Flask 扩展是 Flask-MongoEngine，这个扩展是 MongoEngine 在 Flask 上的扩展，也就是说，我们完全可以独立使用 MongoEngine 而不依赖于 Flask，但依不依赖相差不多，我个人觉得最大的区别在于配置如何处置，所以这里使用依赖 Flask 的扩展。

要在 Flask 中使用 MongoEngine，首先我们需要先在 Flask 中配置 MongoDB 的信息，然后再使用我们的服务器初始化 MongoEngine，这样我们就将数据库和服务器建立了联系，这个在代码中可以这样来表示：

```
app.config['MONGODB_SETTINGS'] = {
    'db': 'the_way_to_flask',
    'host': 'localhost',
    'port': 27017
}

db = MongoEngine()
db.init_app(app)
```

建立联系之后，我们就可以使用 MongoEngine 创建数据模型了。

我们这里还是继承上一章中的数据模型，也就是只有两个字段，分别是 name 和 email：

```
class User(db.Document):
    name = db.StringField()
    email = db.StringField()
```

这样，我们的数据模型创建好了，整段完整的代码是：

```
#!/usr/bin/env python
# encoding: utf-8
from flask import Flask
from flask_mongoengine import MongoEngine


app = Flask(__name__)
app.config['MONGODB_SETTINGS'] = {
    'db': 'the_way_to_flask',
    'host': 'localhost',
    'port': 27017
}

db = MongoEngine()
db.init_app(app)


class User(db.Document):
    name = db.StringField()
    email = db.StringField()


if __name__ == "__main__":
    app.run(debug=True)
```

操作数据

现在我们已经有数据模型(Model)和数据库关联起来了，那光有关联没用啊，我们没办法操作啊。接下来的内容就是讲解如何通过 Model 对数据库中的数据进行增删改查。

查询

MongoEngine 的增删改查非常简单，例如查询，我们可以使用：

```
User.objects(name="zhangsan").first()
```

这个语句就将数据库中名字为 zhangsan 的用户查询出来了。我们来分析一下这个语句是怎么查询的。

首先是 `User.objects` ，这里的 `User` 我们已经知道了是我们的 Model，那既然 User 都已经是 Model 了为什么还要 objects 呢？

就是因为 User 是 Model，因为 Model 本身只代表数据结构，那和我们查询有什么关系呢？所以这里引入了一个 objects 属性，表示一个查询集，这个集合默认就表示 User 表中的所有数据，所以我们后面的 `name=“zhangsan”` 就有点好理解了，其实就是从 User 表中的所有数据中过滤出 name 的值为 zhangsan 的记录，别忘了，过滤出来的数据是一个集合，而不是一个 User 对象，所以我们后面还加了一个 `first` 获取这个集合的第一个元素。

这样，我们就查询到了一个 User 对象。

新增

增加新记录就更简单了，例如我想插入一个 **name** 为 `lisi` ，**email** 为 `lisi@gmail.com` 的用户，那么我们可以这样写：

```
User(name='lisi', email='lisi@gmail.com').save()
```

就这么简单，首先，我们想创建了一个 User 对象，然后调用 save 方法就可以了。

删除

考虑一下如果我们要删除一个记录，我们是不是需要先找到这个需要删除的记录？在 MongoEngine 中就是这样的，如果我们要删除一个记录，我们想找到它，使用的是查询：

```
user = User.objects(name="zhangsan").first()
```

找到之后，很简单，只需调用 delete 方法即可：

```
user.delete()
```

这样，我们就将 zhangsan 这个用户删除掉了。

更新

和删除一样，如果我们需要更新一条记录，那么我们也先需要找到他，假设我们需要更新 lisi 的邮箱为： lisi@outlook.com，那么我们可以这么写：

```
user = User.objects(name="zhangsan").first()
user.update(email="lisi@outlook.com")
```

第一句还是查询啦，第二句这里使用了 update 方法，直接将需要修改的属性以及改变后的值作为参数传入，即可完成更新操作。

完整代码

这样，我们就知道了如何利用模型进行增删改查，那么我们就将这个知识都应用到我们的 REST 服务中，改写后的代码如下：

```
#!/usr/bin/env python
# encoding: utf-8
import json
from flask import Flask, request, jsonify
from flask_mongoengine import MongoEngine


app = Flask(__name__)
app.config['MONGODB_SETTINGS'] = {
    'db': 'the_way_to_flask',
    'host': 'localhost',
    'port': 27017
}

db = MongoEngine()
db.init_app(app)


class User(db.Document):
    name = db.StringField()
    email = db.StringField()

    def to_json(self):
        return {"name": self.name,
                "email": self.email}


@app.route('/', methods=['GET'])
def query_records():
    name = request.args.get('name')
    user = User.objects(name=name).first()

    if not user:
        return jsonify({'error': 'data not found'})
    else:
```

```
        return jsonify(user.to_json())


@app.route('/', methods=['PUT'])
def create_record():
    record = json.loads(request.data)
    user = User(name=record['name'],
                email=record['email'])
    user.save()
    return jsonify(user.to_json())


@app.route('/', methods=['POST'])
def update_record():
    record = json.loads(request.data)
    user = User.objects(name=record['name']).first()
    if not user:
        return jsonify({'error': 'data not found'})
    else:
        user.update(email=record['email'])
    return jsonify(user.to_json())


@app.route('/', methods=['DELETE'])
def delte_record():
    record = json.loads(request.data)
    user = User.objects(name=record['name']).first()
    if not user:
        return jsonify({'error': 'data not found'})
    else:
        user.delete()
    return jsonify(user.to_json())


if __name__ == "__main__":
    app.run(debug=True)
```

CRUD 使用的基本上都是我们介绍的方法，大家可以自己尝试得编写一些。

使用 Flask-Login 注册登录

在我们的前几章中，围绕着要讲解的内容持续得再丰富一个 REST 服务。但是，截止到目前为止，我们这个 REST 服务都是没有权限控制的，也就是说，如果将这个 REST 服务发布到外网上去，那么将可以被任何人操作，增删改查都不是问题。

作为我们的重要服务（真的很重要:-D），我们怎么能让别人随便操作我们的数据呢，所以这一章就讲解一下如何使用 Flask 的又一扩展 **Flask-Login** 来进行访问控制。

安装 Flask-Login

根据在《本书概述》中列举的那样，我们使用的 **Flask-Login** 的版本是

```
Flask-Login==0.3.2
```

所以安装的话直接使用 pip 安装即可：

```
pip install Flask-Login==0.3.2
```

初始化 Flask-Login

和我们在上一章使用 Flask-MongoEngine 一样，使用 Flask-Login 还是依赖于 Flask，所以我们还是需要和 app 这样服务器绑定起来，所以我们一开始还是需要这样和服务器绑定的：

```
from flask.ext.login import LoginManager
login_manager = LoginManager()
login_manager.init_app(app)
```

这样就将 Flask-Login 和服务器绑定起来了。但是，这好像没有什么作用啊，我们要怎么登陆呢？Flask-Login 怎么才知道登录的 URL 的是哪个？怎么验证我们的账号密码？怎么才能知道登陆的用户是谁？这些都是关键的问题啊。

设置 Flask-Login

对于前面提到的问题，我们一一解决，解决完之后我们的 Flask-Login 就差不多算是会使用了。

首先是登陆的 URL 是什么？这个在 Flask-Login 中是没有默认的登陆 URL 的，所以需要我们指定：

```
from flask.ext.login import login_user

login_manager.login_view = 'login'

@app.route('/login', methods=['POST'])
def login():
    info = json.loads(request.data)
    username = info.get('username', 'guest')
    password = info.get('password', '')

    user = User.objects(name=username,
                        password=password).first()
    if user:
        login_user(user)
        return jsonify(user.to_json())
    else:
        return jsonify({"status": 401,
                        "reason": "Username or Password Error"})
```

这里其实就做了两件事：

1. 指定了 login_view 为 'login'
2. 编写的登陆的代码逻辑

那我们来看第一点，指定 login_view，也就是告诉 Flask 我们的处理的登陆的 URL 是哪个。这里我们发现是 'login'，那么 Flask 是怎么根据 login 找到我们的登陆逻辑所在的位置的呢？这里除了 'login' 我们还能填写其他的字符串吗？

这里先给出答案，是不能的，也就是说，在我们这段代码中，必须指定为 'login'，这里的 'login' 的意思就是在当前文件找到

```
def login(self, xxx)
```

这个函数，然后它就是我们处理登陆逻辑代码所在的地方。

假如说我们处理登陆逻辑的代码没有放在这个文件，而是放在了其他文件，例如 auth.py 里面的 login 函数里面，那么我们就需要指定为：

```
login_view = 'auth.login'
```

登陆逻辑

还是看回上一段代码，我们发现这是一个普通的 Flask 处理请求的函数，说普通在于：

- 从客户端的请求中获得参数，和之前的 CRUD 一样
- 无论是登陆成功还是失败都返回 json 串给客户端

那么凭什么这段代码就能胜任登陆用户的职责呢？问题的关键就在于

```
login_user(user)
```

这一句，仅仅是通过这简单的一句，就将当前用户的状态设置成已登录。这里不做过深入的讲解，只需要知道当这个函数被调用之后，用户的状态就是登陆状态了。

那现在问题是，下次有请求过来，我们怎么知道是不是有用户登陆了，怎么知道是哪个用户？这时我们就会发现我们的 Model 还不够完善，需要完善一下 Model。具体应该这样完善一下：

```
class User(db.Document):
    name = db.StringField()
    password = db.StringField()
    email = db.StringField()


    def to_json(self):
        return {"name": self.name,
                "email": self.email}

    def is_authenticated(self):
        return True

    def is_active(self):
        return True

    def is_anonymous(self):
        return False

    def get_id(self):
        return str(self.id)
```

我们可以看到，这里增加了两个方法，分别是：

- is_authenticated：当前用户是否被授权，因为我们登陆了就可以操作，所以默认都是被授权的
- is_anonymous: 用于判断当前用户是否是匿名用户，很明显，如果这个用户登陆了，就必须不是
- is_active： 用于判断当前用户是否已经激活，已经激活的用户才能登陆
- get_id： 获取改用户的唯一标示

这里，我们仅仅可以通过 is_authenticated 来判断用户时候有权限操作我们的 API，但是，我们还不能知道当前的登陆用户是谁，所以我们还需要告诉 Flask-Login 如何通过一个 id 获取到用户的方法：

```
@login_manager.user_loader
def load_user(user_id):
    return User.objects(id=user_id).first()
```

通过指定 user_loader，我们就可以查询到当前的登陆用户是谁了。这样我们就将登陆、判断用户是否登陆都完善起来了。

登陆可见

既然都登陆了，我们就需要控制登陆的权限了，我们设置增加、删除和修改的 REST API 为登陆才能使用，唯有查询的 API 才能随便可见。

控制登陆可用的方法比较简单，只需要加一个 login_required 的装饰器即可。我们还是以之前那些章节的 REST DEMO 为例进行改写：

```
from flask.ext.login import login_required

@app.route('/', methods=['PUT'])
@login_required
def create_record():
    ......

@app.route('/', methods=['POST'])

@login_required
def update_record():
    ......

@app.route('/', methods=['DELETE'])
@login_required
def delte_record():
    ......
```

这样我们就限制了增加、修改和删除操作必须登陆用户才能操作，而我们也能记录是哪个用户做的操作了。

用户信息

既然服务器提供了登陆的支持，那么肯定少不了退出登陆的支持；同时，作为客户端，可能关注的是想知道到底有没有登陆？

对于退出登陆，很简单，都根本不需要使用到 User 的这个 Model 了。代码如下：

```
from flask.ext.login import logout_user

@app.route('/logout', methods=['POST'])
def logout():
    logout_user()
    return jsonify(**{'result': 200,
                      'data': {'message': 'logout success'}})
```

这里就调用了一个 `logout_user` 的方法就退出了登陆。

然而即使退出了登陆客户端也不知道，除非尝试请求一下新增、修改或者删除的操作，发现无法操作了，这时就知道了我已经退出登陆了，这样明显不合理！所以，这里再增加一个获取当前登陆用户信息的接口：

```
from flask.ext.login import current_user

@app.route('/user_info', methods=['POST'])
def user_info():
    if current_user.is_authenticated:
        resp = {"result": 200,
                "data": current_user.to_json()}
    else:

        resp = {"result": 401,
                "data": {"message": "user no login"}}
    return jsonify(**resp)
```

这里一个重要的点就是第一句，这里有一个成员叫做 `current_user` ，这个变量表示的是当前请求的登陆用户，如果登陆了，那么它就是我们设置的 Model User 的对象，根据我们的 Model 定义， is_authenticated 一直为 True，表示登陆了；如果没有登陆，那么它就是默认的匿名用户 AnonymousUserMixin 的对象，is_authenticated 就为 False，就表示没有登陆。

如果登陆的话，那么 current_user 就是 User 的对象了，那么 to_json 方法就可以返回当前登陆用户的用户信息了，这样的话，我们就可以编写获取用户信息的 API 了。

本章的完整代码为：

```
#!/usr/bin/env python
# encoding: utf-8
import json
from flask import Flask, request, jsonify
from flask.ext.login import (current_user, LoginManager,
                             login_user, logout_user,
                             login_required)
from flask_mongoengine import MongoEngine
```

```
app = Flask(__name__)
app.config['MONGODB_SETTINGS'] = {
    'db': 'the_way_to_flask',
    'host': 'localhost',
    'port': 27017
}
app.secret_key = 'youdontknowme'

db = MongoEngine()
login_manager = LoginManager()
db.init_app(app)
login_manager.init_app(app)


login_manager.login_view = 'login'


@login_manager.user_loader
def load_user(user_id):
    return User.objects(id=user_id).first()


@app.route('/login', methods=['POST'])
def login():
    info = json.loads(request.data)
    username = info.get('username', 'guest')
    password = info.get('password', '')

    user = User.objects(name=username,
                        password=password).first()
    if user:
        login_user(user)
        return jsonify(user.to_json())
    else:
        return jsonify({"status": 401,
                        "reason": "Username or Password Error"})


@app.route('/logout', methods=['POST'])
def logout():
    logout_user()
    return jsonify(**{'result': 200,
                      'data': {'message': 'logout success'}})


@app.route('/user_info', methods=['POST'])
def user_info():
    if current_user.is_authenticated:
        resp = {"result": 200,
                "data": current_user.to_json()}
    else:
        resp = {"result": 401,
```

```
                "data": {"message": "user no login"}}
    return jsonify(**resp)


class User(db.Document):
    name = db.StringField()
    password = db.StringField()
    email = db.StringField()

    def to_json(self):
        return {"name": self.name,
                "email": self.email}

    def is_authenticated(self):
        return True

    def is_active(self):
        return True

    def is_anonymous(self):
        return False

    def get_id(self):
        return str(self.id)


@app.route('/', methods=['GET'])
def query_records():
    name = request.args.get('name')
    user = User.objects(name=name).first()

    if not user:
        return jsonify({'error': 'data not found'})
    else:
        return jsonify(user.to_json())


@app.route('/', methods=['PUT'])
@login_required
def create_record():
    record = json.loads(request.data)
    user = User(name=record['name'],
                password=record['password'],
                email=record['email'])
    user.save()
    return jsonify(user.to_json())


@app.route('/', methods=['POST'])
@login_required
def update_record():
    record = json.loads(request.data)
    user = User.objects(name=record['name']).first()
```

```
    if not user:
        return jsonify({'error': 'data not found'})
    else:
        user.update(email=record['email'],
                    password=record['password'])
    return jsonify(user.to_json())


@app.route('/', methods=['DELETE'])
@login_required
def delte_record():
    record = json.loads(request.data)
    user = User.objects(name=record['name']).first()
    if not user:
        return jsonify({'error': 'data not found'})
    else:
        user.delete()
    return jsonify(user.to_json())


if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

自建装饰器实现权限控制

在上一章 《登陆注册》中，我们为 REST 的 API 设置了新增、更新和删除的操作需要登陆才能完成。细想一下，这样未免太过草率，因为对于一个系统来说，用户肯定是分为不同的级别的，例如普通的用户也就只能查查数据，然后一些用户还能多一个增加数据的权限，再高级一点的还能修改数据，最高级的就是增删改查都能。

对于这些更加丰富的需求，我们目前的登陆可用明显还不能满足需求，因此，按常规本章应该会引入一个新的扩展，而 Flask 确实是有一款叫做 `Flask-Principal` ，的扩展可以满足我们的需求，通过这个扩展，我们希望能够达到更细粒度得控制用户的权限。但是，我嫌弃这个扩展太累赘了，所以本章不准备使用这个扩展，而是自己编写一个权限控制的扩展进行权限的控制。

权限控制设计

我们这里的权限控制采用 RBAC 的方式，首先，我们会创建一个 Role 的 Model，然后给每个 User 分配一个 Role，这样的话，我们就可以限制某个操作需要某种 Role 才能执行，这样的话就实现了更细粒度的权限控制。

这里还有个实现细节需要先说明一下，我们的 Role 的权限是以二进制位来表示的，每一个二进制位表示一种权限：

- 第一位表示可以读取记录
- 第二位表示可以新建记录
- 第三位表示可以更新记录
- 第四位表示可以删除记录

这样的话，如果一个用户只能读取记录，那么他对应的 Role 的权限应该是 **0000 0001b** ，换算成十六进制的话就是： 0x01

如果一个用户所有操作都可以执行，那么它的权限应该对应于 **0000 1111b**，换算成十六进制的话就是：0x0f

那么，假如我们要判断一个用户时候可以进行新建操作，那么应该怎么实现这个逻辑？我这里的实现机制是如果是只有新建操作，那么对应的权限就是：**0000 0010b**，那如果我要判断一个用户时候有新建的权限，那么我只需要对这个用户的权限和这个操作所需要的权限进行 **and** 操作，如果得到的结果等于需要的权限的话，那么就表示该用户拥有权限，可能说得有点复杂，上一个简单的例子

```
用户 A 的权限：       0000 0001b       只有读取记录的权限
用户 B 的权限：       0000 1111b       拥有所有权限
新建记录需要权限：    0000 0010b       需要新建权限
用户A是否可以新建：   0000 0001b and 0000 0010b = 0000 0000b ！= 新建权限，所以不能新建
用户B时候可以新建：   0000 1111b and 0000 0010b = 0000 0010b  == 新建权限，所以可以新建
```

大概就是这样一个场景，大家可以自己动手演练演练，看下是否可行。

创建 Role Model

之前已经在《集成数据库》章节中讲解过了如何创建 Model，所以这里直接根据之前的经验创建 Role Model，然后再往 User 中加上一个 Role 字段。

```
class Permission:
    READ = 0x01
    CREATE = 0x02
    UPDATE = 0x04
    DELETE = 0x08
    DEFAULT = READ

class Role(db.Document):
    name = db.StringField()
    permission = db.IntField()

class User(db.Document):
    name = db.StringField()
    password = db.StringField()
    email = db.StringField()
    role = db.ReferenceField('Role'， default=DEFAULT_ROLE)
```

这里就简单得创建了一个 Role 的 Model，而 Role 只有一个名称，用于标示这个角色，另外一个就是该角色拥有的权限了。然后就是在 User 中添加了一个 ReferenceField，这个在 MongoEngine 里面就表示是外引用的意思，我们可以直接通过这个成员变量访问到用户的 Role 的 permission。

同时，为了保持代码的可维护性，我们将 permission 都写在一个类中，还设置了一个默认的权限，默认为 READ。

因为我们现在的数据库中还没有 Role 相关的记录，所以我们需要在启动应用的时候进行插入数据，所以我做了这样的一个操作：

```
# init roles
if Role.objects.count() <= 0:
    READ_ROLE = Role('READER', Permission.READ)
    CREATE_ROLE = Role('CREATER', Permission.CREATE)
    UPDATE_ROLE = Role('UPDATER', Permission.UPDATE)
    DELETE_ROLE = Role('DELETER', Permission.DELETE)
    DEFAULT_ROLE = Role('DEFAULT', Permission.DEFAULT)

    READ_ROLE.save()
    CREATE_ROLE.save()
    UPDATE_ROLE.save()
    DELETE_ROLE.save()
    DEFAULT_ROLE.save()
else:
    READ_ROLE = Role.objects(permission=Permission.READ).first()
    CREATE_ROLE = Role.objects(permission=Permission.CREATE).fir
st()
    UPDATE_ROLE = Role.objects(permission=Permission.UPDATE).fir
st()
    DELETE_ROLE = Role.objects(permission=Permission.DELETE).fir
st()
    DEFAULT_ROLE = Role.objects(permission=Permission.DEFAULT).f
irst()
```

虽然这段代码有不严谨的地方，但是作为讲解的话无关大雅，通过这段代码，我们可以保证在下面的代码中我们有五种 Role 的对象，分别对应着增删改查，还有一个默认的角色，他为读取权限。同时，我们也应该修改一下我们的 API，让他能够增加用户的默认权限。

```
@app.route('/', methods=['POST'])
@login_required
def create_record():
    record = json.loads(request.data)
    user = User(name=record['name'],
                password=record['password'],
                email=record['email'],
                role=DEFAULT_ROLE)
    user.save()
    return jsonify(user.to_json())
```

这段代码只增加了一行，就是：

```
role=DEFAULT_ROLE
```

权限控制

好，到这里算是完成了一半了，我们的角色已经算是有了，然后就是怎么进行权限控制了，我希望权限控制代码能够竟可能得简单，最好是能用装饰器实现，对于一些默认权限就能访问的，我希望不用加权限控制的代码就好了。没有不能实现的需求，只是实现得好坏而已，所以，既然我们都能描述出需求，那么就能够写出满足需求的代码。

首先，我们是需要编写一个权限控制的装饰器的，我们希望这个装饰器可以很方便得进行权限控制，最好是可以这样：

```
@creater_required()
def create_model():
    ... ...
```

或者这样也可以接受：

```
@permission_required(CREATE_PERMISSION):
def create_model():
    ... ...
```

那么，就先写一个较为简单的版本试试先：

```
def permission_required(permission):
    def decorator(func):
        @wraps(func)
        def decorated_function(*args, **kwargs):
            if not current_user.is_authenticated:
                abort(401)
            user_permission = current_user.role.permission
            if user_permission & permission == permission:
                return func(*args, **kwargs)
            else:
                abort(403)
        return decorated_function
    return decorator
```

这一版本我们可以简单得看这几句关键的代码：

```
if not current_user.is_authenticated:
    abort(401)
user_permission = current_user.role.permission
if user_permission & permission == permission:
    return func(*args, **kwargs)
else:
    abort(403)
```

首先用户没有登陆肯定是没有权限的了，所以返回 401 未授权错误，如果用户没有权限（权限设计中的描述），那么就返回 403 禁止访问。

接着我们就在我们的 REST API 中尝试一下这个权限，这里相对新增用户进行尝试：

```
@app.route('/', methods=['POST'])
@permission_required(Permission.CREATE)
def create_record():
    record = json.loads(request.data)
    user = User(name=record['name'],
                password=record['password'],
                email=record['email'],
                role=DEFAULT_ROLE)
    user.save()
    return jsonify(user.to_json()
```

这里只将 `@login_required` 的装饰器换成了

```
@permission_required(Permission.CREATE)
```

然后我们尝试一下新建记录：

```
POST http://localhost:8080

{
  "email": "liqianglau@outlook.com",
  "name": "tyrael",
  "password": "password"
}
```

然后发现响应是：

Forbidden

You don't have the permission to access the requested resource. It is either read-protected or not readable by the server.

说明我们的权限控制生效啦。

规范结构维护代码

到本章为止，我们的 DEMO 程序功能已经日益强大，增删改查，用户登录，权限控制，数据库操作，功能已经有点复杂了，然后看看代码，发现也已经差不多 200 行了。这时，我们不禁要想，难道我们要在这一个 `app.py` 文件中继续编写下去吗？感觉每次添加新的功能好像都是在头(添加引用)在尾(添加逻辑)添加代码，难道这是正确的做法吗？

很显然，作为有洁癖的工程师，肯定不能容忍所有代码都这么一团塞在一个文件里面的，所以我们至少会想到拆分代码然后放到几个模块里面，例如什么 model.py, controller.py 啊之类的。但是，作为有深度洁癖的人，觉得把一大堆文件放在一个目录里面也是件糟心的事，所以，这里我要介绍一下我比较推荐的代码目录结构。

Flask 代码目录结构

虽然目录结构见仁见智，个人有个人的看法和习惯，但总的来说，经过很多人的实践和总结，还是有很多共同的意见和想法的，而我在查看他人的目录结构结合自身在工作中的使用经验，总结了一个个人认为比较恰当的目录结构供参考，而本书也是以这个目录结构为架构进行下去的。

我推荐的目录结构：

```
.
├── README.md
├── application
│   ├── __init__.py
│   ├── controllers
│   │   └── __init__.py
│   ├── forms
│   │   └── __init__.py
│   ├── models
│   │   └── __init__.py
│   ├── services
│   │   └── __init__.py
│   ├── static
│   │   └── __init__.py
│   ├── templates
│   │   └── __init__.py
│   └── utils
│       └── __init__.py
├── config
│   ├── __init__.py
│   ├── default.py
│   ├── development.py
│   ├── development_sample.py
│   ├── production.py
│   ├── production_sample.py
│   └── testing.py
├── deploy
│   ├── flask_env.sh
│   ├── gunicorn.conf
│   ├── nginx.conf
│   └── supervisor.conf
├── manage.py
├── pylintrc
├── requirements.txt
├── tests
│   └── __init__.py
└── wsgi.py
```

这里稍作介绍，首先是第一级目录的话，主要分为两类，一类是目录，另一类是运行相关的文件；其中目录有：

- application：项目所有逻辑代码都放这
- config：项目的配置文件，按不同环境各占一份
- deploy：部署相关的文件，后续将使用到
- tests：单元测试代码所在的目录

文件的话分别有：

- manage.py：Flask-Script 运行文件，后面介绍
- pylintrc：静态分析代码使用的 pylint 标准
- requirements.txt：项目依赖库的列表

- wsgi.py：wsgi 运行的文件

规范代码到指定目录

既然我们已经规定好了目录结构，是时候将我们的意面分到各个盘子里了。首先从文件开始，因为我们还没有介绍 Flask-Script，静态检查和 wsgi，所以就忽略这些文件，那么就剩下 requirements.txt 文件了。这个文件的内容都在我们的《本书概述》中列举了，直接放进去就好了。

```
Flask==0.10.1
flask-mongoengine==0.7.5
Flask-Login==0.3.2
Flask-Admin==1.4.0
Flask-Redis==0.1.0
Flask-WTF==0.12
```

然后是时候解耦代码了，我们没有表单，暂时没有 services，没有静态文件也没有页面模板，所以可以这样合并：

- 将 route 代码放到 application/controllers 中
- 将 model 代码放到 application/models 中
- 将初始化绑定 app 的代码放到 application/**init**.py 中
- 将 数据库等配置放到 config/development.py 中

最后就是编写 manager.py 文件了。这里概要得列举几个重要的文件，更多的文件大家可以从 github 上 clone 代码出来阅读。

合并后的文件

manager.py

```
# coding: utf-8
from flask.ext.script import Manager
from application import create_app

# Used by app debug & livereload
PORT = 8080

app = create_app()
manager = Manager(app)


@manager.command
def run():
    """Run app."""
    app.run(port=PORT)


if __name__ == "__main__":
    manager.run()
```

application/**init**.py

```
# coding: utf-8
import sys
import os

# Insert project root path to sys.path
project_path = os.path.abspath(os.path.join(os.path.dirname(__fi
le__), '..'))
if project_path not in sys.path:
    sys.path.insert(0, project_path)

import logging
from flask import Flask
from flask_wtf.csrf import CsrfProtect
from config import load_config
from application.extensions import db, login_manager
from application.models import User
from application.controllers import user_bp

# convert python's encoding to utf8
try:
    reload(sys)
    sys.setdefaultencoding('utf8')
except (AttributeError, NameError):
    pass


def create_app():
    """Create Flask app."""
    config = load_config()
```

```
    print config

    app = Flask(__name__)
    app.config.from_object(config)

    if not hasattr(app, 'production'):
        app.production = not app.debug and not app.testing

    # CSRF protect
    CsrfProtect(app)

    if app.debug or app.testing:
        # Log errors to stderr in production mode
        app.logger.addHandler(logging.StreamHandler())
        app.logger.setLevel(logging.ERROR)

    # Register components
    register_extensions(app)
    register_blueprint(app)

    return app


def register_extensions(app):
    """Register models."""
    db.init_app(app)
    login_manager.init_app(app)

    login_manager.login_view = 'login'

    @login_manager.user_loader
    def load_user(user_id):
        return User.objects(id=user_id).first()


def register_blueprint(app):
    app.register_blueprint(user_bp)
```

application/controllers/**init**.py

```
#!/usr/bin/env python
# encoding: utf-8
import json

from flask import Blueprint, request, jsonify
from flask.ext.login import current_user, login_user, logout_use
r

from application.models import User


user_bp = Blueprint('user', __name__, url_prefix='')


@user_bp.route('/login', methods=['POST'])
def login():
    info = json.loads(request.data)
    username = info.get('username', 'guest')
    password = info.get('password', '')

    user = User.objects(name=username,
                        password=password).first()
    if user:
        login_user(user)
        return jsonify(user.to_json())
    else:
        return jsonify({"status": 401,
                        "reason": "Username or Password Error"})


@user_bp.route('/logout', methods=['POST'])
def logout():
    logout_user()
    return jsonify(**{'result': 200,
                      'data': {'message': 'logout success'}})

@user_bp.route('/user_info', methods=['POST'])
def user_info():
    if current_user.is_authenticated:
        resp = {"result": 200,
                "data": current_user.to_json()}
    else:
        resp = {"result": 401,
                "data": {"message": "user no login"}}
    return jsonify(**resp)
```

config/development.py

```
# coding: utf-8
import os


class DevelopmentConfig(object):
    """Base config class."""
    # Flask app config
    DEBUG = False
    TESTING = False
    SECRET_KEY = "sample_key"

    # Root path of project
    PROJECT_PATH = os.path.abspath(os.path.join(os.path.dirname(
__file__), '..'))

    # Site domain
    SITE_TITLE = "twtf"
    SITE_DOMAIN = "http://localhost:8080"

    # MongoEngine config
    MONGODB_SETTINGS = {
        'db': 'the_way_to_flask',
        'host': 'localhost',
        'port': 27017
    }
```

建立目录管理配置

在前面一章《更好得维护代码》 中，我们将项目按照功能作用划分到不同的目录中，这样使得我们的项目结构更加清晰和规整了。但是，因为上一章节的内容比较多，如果作为初学者来说，肯定是有好多有疑问的地方，从本章开始都会进行介绍，让大家对 Flask 的使用更加得得心应手。

本章主要介绍的是 Flask 中的配置管理，从前面章节《更好得维护代码》 里，可以发现，配置目录 **config** 下包含多个配置文件，为什么要包含这么多文件，而我们要如何处理这些配置文件，都是本章的讲解内容。

```
├── config
│   ├── __init__.py
│   ├── default.py
│   ├── development.py
│   ├── development_sample.py
│   ├── production.py
│   ├── production_sample.py
│   └── testing.py
```

环境分类

有一个重要的概念是需要我们关注的，那就是每个配置文件都是与环境相关的，也就是说，就是因为有多个环境，所以才会出现多个配置。如果不太理解这句话的意思的话，我们看一下 config 目录下的文件名，其实可以分为几类：

- development： 开发环境，一般为本地开发环境使用
- production：生产环境，一般为线上部署运行环境使用
- testing： 测试环境，一般用于各种测试使用

如我们所看到的一样，我们在平时的工作中会有各种环境，我们在本地开发调试的时候应该有个本地环境，当我们转测试之后后会有个测试环境，测试完成之后放到线上之后会有个线上正式环境，而每个环境很难保持配置完全一致，所谓的不一致是指例如数据库信息、程序运行的模型等，例如我们本地的开发环境数据库地址是：

```
MongoDB：
version：3.2.6
ip：localhost
port：27017
```

而在生产环境却是：

```
MongoDB：
    version：3.2.6
    ip：192.168.59.104
    port：27017
```

所以为了方便开发、测试和部署，我们就会设置多份配置文件，这样就可以快速得在不同环境中运行。如果你有其他的情况，可以随时添加配置文件，完全没问题。

加载配置

那这么多份配置文件，我如何让程序制定加载哪份配置文件呢？这里的奥妙就在 **`config/__init__.py`** 文件中。我们打开这个文件看看：

```
# coding: UTF-8
import os


def load_config(mode=os.environ.get('MODE')):
    """Load config."""
    try:
        if mode == 'PRODUCTION':
            from .production import ProductionConfig
            return ProductionConfig
        elif mode == 'TESTING':
            from .testing import TestingConfig
            return TestingConfig
        else:
            from .development import DevelopmentConfig
            return DevelopmentConfig
    except ImportError:
        from .default import Config
        return Config
```

在 config/**init**.py 文件中，我定义了一个 load_config 函数，这个函数接受一个 `mode` 参数，表示是获取什么环境的配置，如果不传这个参数的话，那默认使用的就是系统环境变量中的 `MODE` 环境变量，然后就根据指定的环境返回指定的配置文件。

如果没有指定的配置文件的话，那么就只能返回默认环境变量了。同样的，如果你需要新增自定义的环境配置文件，那么只需要简单得修改这个函数，并且指定加载你自定义的配置文件即可。

使用配置

加载配置这一问题解决之后，接下来就是在我们的 Flask 应用中使用这些配置了，既然都 load 好了配置，那么使用也就问题不大了，这里是一个示例：

```
"""Create Flask app."""
config = load_config(mode)

app = Flask(__name__)
app.config.from_object(config)
```

这里首先将配置 load 出来，然后使用 Flask 对象的 config.from_object 设置配置。就这么简单。

总结

本章对 Flask 中如何配置多环境的配置文件进行了说明和介绍，然后分析了如何加载不同配置文件的原理，最后，给出了一个如何在实际应用中使用配置的示例。

使用 Flask-Script 启动应用

看到这章的内容也许你会有疑惑，启动应用？不是很简单吗？我直接使用

```
python app.py
```

不就将应用跑起来了吗，而且我还能看到访问的日志呢。是的，没错，直接运行代码是可以将我们编写的 Web 应用跑起来，而且还能很好得查看运行信息，但是，假设你想更换配置呢？例如，你有 development1.py 和 development2.py 两个配置文件，一开始你使用 development1.py 运行，然后你想换成 development2.py 这个配置文件，那么你需要怎么做？根据我们在《配置管理》中介绍的那样，你有两个选择，分别是：

- 修改系统变量 MODE
- 修改代码，直接指定 create_app('development1') => create_app('development2')

看上去都不是很方便，因为这至少涉及到两个动作，第一个是修改模式，第二个是启动应用。那么这个时候，假如我们在运行应用的时候能够指定需要使用的配置文件的话，那不是方便多了，例如：

```
python app.py development1
```

这样的话，好像就好多了，确实，这样确实满足了我们的需求，但是，这仅仅满足了一个需求，那万一我们还想看我们的应用对外暴露的 API 有哪些呢？我还想使用我们应用的 python shell 呢？这些都是比较难实现的。然而，作为一个提供丰富扩展的框架，Flask 的贡献者们也已经为我们想到了，并且给我们提供了一个扩展 Flask-Script，可以让我们从这些繁琐的事情中解放出来。

可能机智的你已经发现了，在我们的《更好得维护代码》中已经根目录里面多了一个 manage.py 的文件，是的，这个文件就是为使用 Flask-Script 而创建的，而我们启动应用也将使用这个文件。下面就来介绍一下 Flask-Script 的一些知识。

安装 Flask-Script

依旧还是老套路，直接使用 pip 安装既可。

```
pip install Flask-Script
```

小试身手

和我们之前使用过的 Flask 扩展不一样，Flask-Script 不需要获取我们 app 的配置信息，所以就不用使用 init_app 这样的初始化操作了，但是，毕竟 app 是我们的 Flask 服务器，所以还是需要使用到它，所以我们一开始的启动脚本可以这样写：

```
# coding: utf-8
from flask_script import Manager
from application import create_app

app = create_app('development')
manager = Manager(app)


if __name__ == "__main__":
    manager.run()
```

我们这就做了两个操作，分别是：

```
manager = Manager(app)
manager.run()
```

很奇怪的是，和我们最开始的运行的相比，好像更复杂了，因为我们最初的版本直接这样跑就可以了：

```
app.run()
```

那为什么要多一个 manager 呢？因为manager 可以做更多的操作，例如指定运行参数，查看所有 API 等。

指定运行参数

如果我们想指定配置文件，Flask-Script 提供了一个 -c 的参数，然后可以这样做：

```
#!/usr/bin/env python
# encoding: utf-8
from flask_script import Manager
from application import create_app

manager = Manager(create_app)
manager.add_option('-c', '--config', dest='mode', required=False
)


if __name__ == "__main__":
    manager.run()
```

其实这里做的改变就是不直接创建 app，而是将创建方法直接传给 Manager，然后多了重要的一行，那就是：

```
manager.add_option('-c', '--config', dest='mode', required=False
)
```

通过名字可以发现这是一个添加选项的语句，默认就是给我们的 create_app 添加参数选项，然后我们看一下它的参数分别有哪些：

- -c : 运行参数的简写
- --config : 运行参数的全写
- dest : 传递给 create_app 的参数名字，因为我们是 create_app(mode) ，所以这里是 'mode'
- required : 是否是必须的，这里因为有默认的 mode ，所以不需要必选。

就这样我们就可以给 create_app 传递参数了，那怎么传，这样子：

```
python manage.py -c development  # 开发环境运行
python manage.py -c testing      # 测试环境运行
```

就是这么简单，我们就可以在执行得时候指定要运行的环境了。

查看所有暴露的 API

很多时候，因为我们的项目是多人开发，所以我们经常会不知道我们的代码暴露了哪些 API。事实上，这在多人协作的项目中是非常常见的问题，例如我经历过的两个企业，其中一家是世界500强的 IT 企业，都存在这个问题，而他们的解决方法就是使用一份 Excel 管理暴露的接口，而这些接口都由各个模块的负责人自己填写，当然，后面我开发了一个统一管理系统对 API 进行管理，但毕竟在大的企业中，很难协调好所有的部门和产品，所以还是很难有全局的视角。

但是，在 Flask 中，我们就不需要担心有这种问题了，因为我们的 Flask-Script 还是提供了一个命令可以快速得帮助我们列举出我们的公开接口，使用上也很简单，直接这样写即可：

```
from flask_script.commands import ShowUrls

manager.add_command("showurls", ShowUrls())
```

然后，我们在控制台上敲以下命令：

```
python manage.py showurls
```

你很惊喜得会发现这些输出：

```
Rule                     Endpoint
---------------------------------------
/login                   user.login
/logout                  user.logout
/static/<path:filename>  static
/user_info               user.user_info
```

它将我们所有的公开接口都打印出来了，但是，可能你也发现了，不完善的地方就是这里只给出了 URI，并没有给出请求方法，例如 GET、POST 等。这是有待提高的地方。

总结

本章介绍了 Flask-Script 这一扩展的使用，并且介绍了两个用法，分别是使用指定参数启动应用以及查看所有暴露出来的 API URI，但是也许你会有一些新的想法，但是 Flask-Script 并没有提供给你，没关系，Flask 作为一个对扩展友好的框架，你有任何想法都可以通过扩展来实现，更多的详情读者可以参考Flask-Script官方文档来实现。

使用 **Flask-Admin** 管理数据库数据

我们回过头来看看我们到目前为止的 REST API，发现好像现在都不知道有多少条 User 记录了，甚至于连获取所有 User 记录的 API 都没提供，更别说随便查看用户的记录了。面对这个困境， Flask 的扩展是否还能给我们更多的惊喜呢？答案肯定还是可以的。这一章节，我将带读者认识一个 Flask 中的管理扩展 —— Flask-Admin。

使用 Flask-Admin，我们可以方便快捷得管理我们的 Model 数据，让我们能够省去一大堆开发管理系统的时间，而更多得将精力放到梳理业务逻辑之上，下面就开始讲解一下如何使用 Flask-Admin。

安装 **Flask-Admin**

没有什么特殊的，还是直接使用 pip 安装：

```
pip install Flask-Admin==1.4.0
```

就直接安装上了 Flask-Admin 扩展，然后等待后续使用

初始化 **Flsak-Admin**

和其他常见扩展一般，Flask-Admin 还是需要和我们的 app 服务器绑定，所以还是老套路，但是，因为我们规范化了我们的目录结构，所以这里我们需要注意的是，创建 Flask-Admin 对象要放在 application/extensions.py 文件中，所以在我们的 application/extensions.py 中已经写入以下语句：

```
from flask.ext.admin import Admin
admin = Admin()
```

接下来就是要和我们的 Flask 服务器进行绑定了，还是老套路，不过还是因为规范化的原因，我们的绑定需要放到 **application/init.py** 中执行，那就需要在 **application/init.py** 文件中的 register_extensions 函数中添加以下语句：

```
from application.extensions import admin

admin.init_app(app)
```

然后就算完成了，接下来，我们运行服务器试试，此时我们运行服务器是使用以下语句了：

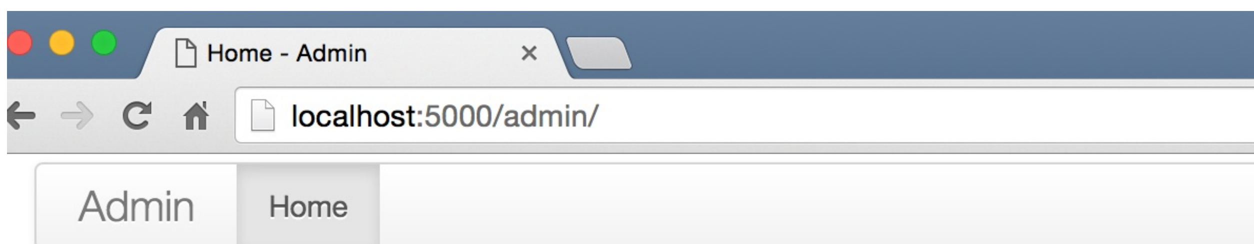
```
python manage.py runserver
```

初见 Flask-Admin

当我们服务器跑起来之后，我们要想看到管理界面，只需要在浏览器中输入URL：

```
http://localhost:5000/admin
```

你就能看到最简单的管理后台了，但是！！这里面什么都没有，就像这样：



看来第一印象不是很好，那么，我们要怎样才能看到东西？其实也不复杂，既然没有数据，那我们就将我们的数据 Model 加进去，怎么加，下面给出一个简单的例子，同样还是在 application/**init**.py 中：

```
from flask_admin.contrib.mongoengine import ModelView

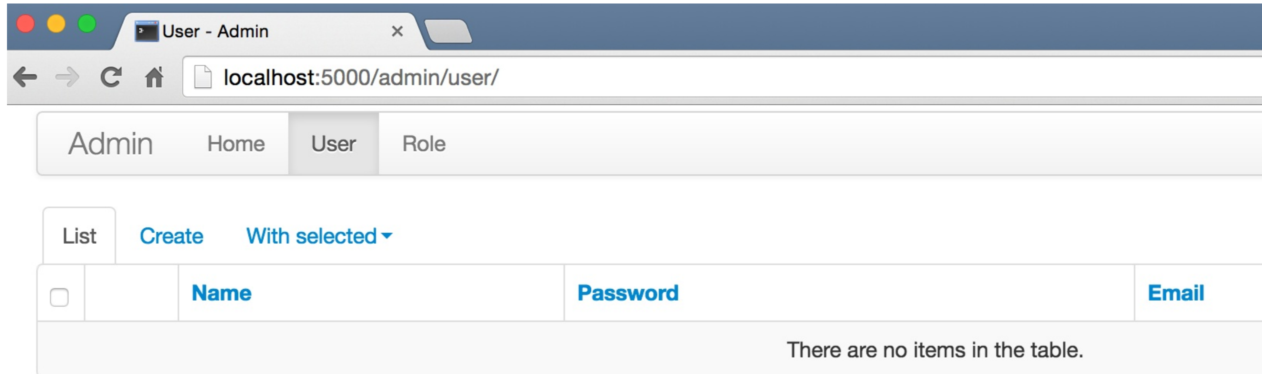
from application.models import User, Role

def register_extensions(app):
    admin.init_app(app)
    admin.add_view(ModelView(User))
    admin.add_view(ModelView(Role))
```

这样就可以了，还是那样，重启一下我们的服务器再访问：

```
http://localhost:5000/admin
```

这个时候，我们会发现有两个 Model 了：



操作 Flask-Admin

在后台中我们可以看到一些选项，例如List、Create、With select，然后这些选项下面是一个表格，也许你会发现这个表格是空的，那是因为你的数据库中没有数据，所以是空的很自然，那么我们要怎么添加数据呢？试试点一下 “Create” 看下：

Admin Home User Role

List Create

name

password

email

role ---

Save | Save and Add Another | Save and Continue Editing | Cancel

看到的是这个，我们填充完各个字段之后，点提交就能看到表格中有数据了。

Admin Home User Role

Record was successfully created.

List (1) Create With selected

		Name	Password	Email	Role
		zhangsan	password	zhangsan@gmail.com	

总结

本章很简约得介绍了 Flask 中的管理扩展 Flask-Admin，并且演示了如何添加管理我们的 Model 数据，并且简单得介绍了一下支持的操作，但是这些都只是皮毛，如果读者有兴趣的话，可以阅读我的文章《Flask-Admin》了解更多知识，也可以查看 Flask-Admin 的官方文档学习关于 Flask-Admin 的内容。

第三部分

Flask 项目实战

- TODO
- TODO
- TODO
- TODO

编写 TODO 应用【part001】

本书前面两个部分分别对 Flask 的基本知识、用法以及介绍了多种扩展以及扩展的通用使用方式，使用扩展过程中的一些细节进行了讲解。虽然过程中有一个 REST API 小例子描述，但是，毕竟是作为各个扩展使用讲解而编排在一块，所以缺乏系统性，全局性。

从本章开始，我将使用 Flask 围绕一个 TODO 应用提供 REST API 进行讲解，让大家有个对 Flask 应用有一个直观的认识。

TODO 应用讲解

我们需要编写的 TODO 应用主要功能有：

- 可以查询所有待办事项
- 可以查看指定待办事项的详情
- 可以增加一项待办事项
- 可以删除一项待办事项
- 可以修改一项待办事项，包括待办内容，添加标记
- 完成待办事项后可以标记为完成

这些就是我们应用的简略需求，然后再讲一下我们的项目结构，根据前面章节《更好得维护代码》中讲解的，我们将项目结构设计成如下：

```
.
├── README.md
├── application
│   ├── __init__.py
│   ├── controllers
│   │   ├── __init__.py
│   │   ├── auth.py
│   │   ├── todo.py
│   │   └── user.py
│   ├── extensions.py
│   └── models
│       ├── __init__.py
│       ├── todo.py
│       └── user.py
├── commands.py
├── config
│   ├── __init__.py
│   ├── default.py
│   ├── development.py
│   ├── development_sample.py
│   ├── production.py
│   ├── production_sample.py
│   └── testing.py
├── deploy
│   ├── flask_env.sh
│   ├── gunicorn.conf
│   ├── nginx.conf
│   └── supervisor.conf
├── manage.py
├── pylintrc
├── requirements.txt
├── tests
│   └── __init__.py
└── wsgi.py
```

设计 Models

Model 的话主要设计两个主要的模型，分别是 User 和 Item。User 表示用户的信息，除了表示TODO 所属人之外，还有登录的用处，而 Item 则是待办事项了，具体设计需要参考需求而定，关于 Model 的具体设计过程不是本章讨论的重点，所以直接给出 Models：

application/models/**init**.py

```
#!/usr/bin/env python
# encoding: utf-8
from user import *
from todo import *


def all():
    result = []
    models = [user, todo]

    for m in models:
        result += m.__all__

    return result

__all__ = all()
```

application/models/user.py

```
#!/usr/bin/env python
# encoding: utf-8
from application.extensions import db

__all__ = ['Role', 'User']


class Permission:
    READ = 0x01
    CREATE = 0x02
    UPDATE = 0x04
    DELETE = 0x08
    DEFAULT = READ


class Role(db.Document):
    name = db.StringField()
    permission = db.IntField()


class User(db.Document):
    name = db.StringField()
    password = db.StringField()
    email = db.StringField()
    role = db.ReferenceField('Role')

    def to_json(self):
        return {"name": self.name,
                "email": self.email,
                "role": self.role.name}

    def is_authenticated(self):
        return True

    def is_active(self):
        return True

    def is_anonymous(self):
        return False

    def get_id(self):
        return str(self.id)
```

application/models/todo.py

```
#!/usr/bin/env python
# encoding: utf-8
from application.extensions import db

__all__ = ['Item']


class Item(db.Document):
    content = db.StringField(required=True)
    created_date = db.DateTimeField()
    completed = db.BooleanField(default=False)
    completed_date = db.DateTimeField()
    created_by = db.ReferenceField('User', required=True)
    notes = db.ListField(db.StringField())
    priority = db.IntField()

    def __repr__(self):
        return "<Item: {} Content: {}>".format(str(self.id),
                                               self.content)

    def to_json(self):
        return {
            'id': str(self.id),
            'content': self.content,
            'completed': self.completed,
            'completed_at': self.completed_date.strftime("%Y-%m-
%d %H:%M:%S") if self.completed else "",
            'created_by': self.created_by.name,
            'notes': self.notes,
            'priority': self.priority
        }
```

设计 views

根据我们在前面章节所学习的知识，我们这个应用的 views 就不是直接使用 `app.route` 来绑定 URL 了，而是使用 Blueprint 来设计，具体设计如下：

application/controller/**init**.py

```
#!/usr/bin/env python
# encoding: utf-8
import auth
import user
import todo

all_bp = [
    auth.auth_bp,
    user.user_bp,
    todo.todo_bp
]
```

application/controller/auth.py

```
#!/usr/bin/env python
# encoding: utf-8
import json

from flask import Blueprint, request, jsonify
from flask.ext.login import login_user, logout_user

import application.models as Models


auth_bp = Blueprint('auth', __name__, url_prefix='/auth')


@auth_bp.route('/login', methods=['POST'])
def login():
    info = json.loads(request.data)
    username = info.get('username', 'guest')
    password = info.get('password', '')

    user = Models.User.objects(name=username,
                                password=password).first()
    if user:
        login_user(user)
        return jsonify(user.to_json())
    else:
        return jsonify({"status": 401,
                        "reason": "Username or Password Error"})


@auth_bp.route('/logout', methods=['POST'])
def logout():
    logout_user()
    return jsonify(**{'result': 200,
                      'data': {'message': 'logout success'}})
```

application/controller/user.py

```
#!/usr/bin/env python
# encoding: utf-8
from flask.ext.login import current_user
from flask import Blueprint, jsonify


user_bp = Blueprint('users', __name__, url_prefix='')


@user_bp.route('/user_info', methods=['POST'])
def user_info():
    if current_user.is_authenticated:
        resp = {"result": 200,
                "data": current_user.to_json()}
    else:
        resp = {"result": 401,
                "data": {"message": "user no login"}}
    return jsonify(**resp)
```

application/controller/todo.py

```
#!/usr/bin/env python
# encoding: utf-8
import json
from datetime import datetime

from flask import Blueprint, request, jsonify
from flask.ext.login import current_user, login_required

import application.models as Models


todo_bp = Blueprint('todos', __name__, url_prefix='/todo')


@todo_bp.route('/item', methods=['POST'])
@login_required
def create_todo_item():
    data = json.loads(request.data)
    content = data.get('content')
    note = data.get('note', None)
    priority = data.get('priority', 0)

    if not content:
        return jsonify({
            'data': {},
            'msg': 'no content',
            'code': 1001,
            'extra': {}})
```

```
    item = Models.Item(content=content, created_date=datetime.no
w(),
                       completed=False, created_by=current_user.
id,
                       notes=[note] if note else [],
                       priority=priority)
    item.save()
    return jsonify({
        'data': item.to_json(),
        'msg': 'create item success',
        'code': 1000,
        'extra': {}
    })


@todo_bp.route('/item', methods=['DELETE'])
@login_required
def delete_todo_item():
    data = json.loads(request.data)
    id = data.get('id')

    if not id:
        return jsonify({
            'data': {},
            'msg': 'no id',
            'code': 2001,
            'extra': {}})

    item = Models.Item.objects(id=id).first()
    item.delete()
    return jsonify({
        'data': item.to_json(),
        'msg': 'delete item success',
        'code': 2000,
        'extra': {}
    })


@todo_bp.route('/item', methods=['PUT'])
@login_required
def update_todo_item():
    data = json.loads(request.data)
    id = data.get('id')
    type = data.get('type')

    if type == "update_content":
        content = data.get('content')
        Models.Item.objects(id=id).first().update(content=conten
t)
    elif type == "insert_notes":
        note = data.get('note')
        Models.Item.objects(id=id).first().update(push__notes=no
```

```
te)
    elif type == "done":
        Models.Item.objects(id=id).first().update(completed=True
,
                                                  completed_date
=datetime.now())
    return jsonify({
        'data': {'oper': type,
                 'id': id},
        'msg': 'oper done',
        'code': 3000,
        'extra': {}
    })


@todo_bp.route('/item', methods=['GET'])
@login_required
def get_todo_item():
    query_string = request.args.get('q')
    data = json.loads(query_string)
    id = data.get('id')

    item = Models.Item.objects(id=id).first()
    return jsonify({
        'data': item.to_json(),
        'msg': 'query item success',
        'code': 4000,
        'extra': {}
    })


@todo_bp.route('/items', methods=['GET'])
@login_required
def get_todo_items():
    data = json.loads(request.args.get('q'))
    page = data.get('page', 1)
    page_size = data.get('page_size', 10)

    begin = (page - 1) * page_size
    end = begin + page_size
    items = Models.Item.objects()[begin: end]
    rsts = []
    for item in items:
        rsts.append(item.to_json())

    return jsonify({
        'data': rsts,
        'msg': 'query items success',
        'code': 5000,
        'extra': {}
    })
```

初始化扩展

扩展我们是统一放到 application/extensions.py 里面进行构建对象的，所以文件有：

application/extensions.py

```
#!/usr/bin/env python
# encoding: utf-8
from flask.ext.admin import Admin
from flask.ext.login import LoginManager
from flask.ext.mongoengine import MongoEngine


db = MongoEngine()
login_manager = LoginManager()
admin = Admin()
```

初始化应用

```
#!/usr/bin/env python
# encoding: utf-8
import sys
import logging

from flask import Flask
from flask_admin.contrib.mongoengine import ModelView

from config import load_config
from application.extensions import db, login_manager, admin
from application.models import User, Role
from application.controllers import all_bp

# convert python's encoding to utf8
try:
    reload(sys)
    sys.setdefaultencoding('utf8')
except (AttributeError, NameError):
    pass


def create_app(mode):
    """Create Flask app."""
    config = load_config(mode)

    app = Flask(__name__)
    app.config.from_object(config)
```

```
    if not hasattr(app, 'production'):
        app.production = not app.debug and not app.testing

    if app.debug or app.testing:
        # Log errors to stderr in production mode
        app.logger.addHandler(logging.StreamHandler())
        app.logger.setLevel(logging.ERROR)

    # Register components
    register_extensions(app)
    register_blueprint(app)

    return app


def register_extensions(app):
    """Register models."""
    db.init_app(app)
    login_manager.init_app(app)

    # flask-admin configs
    admin.init_app(app)
    admin.add_view(ModelView(User))
    admin.add_view(ModelView(Role))

    login_manager.login_view = 'auth.login'

    @login_manager.user_loader
    def load_user(user_id):
        return User.objects(id=user_id).first()


def register_blueprint(app):
    for bp in all_bp:
        app.register_blueprint(bp)
```

编写 TODO 应用【part002】

设置配置

配置的话，我们全放在 config 目录下，并且按环境划分，因为只使用到开发环境，所以就只设置了开发环境的：

config/**init**.py

```
# coding: UTF-8
import os


def load_config(mode=os.environ.get('MODE')):
    """Load config."""
    try:
        if mode == 'PRODUCTION':
            from .production import ProductionConfig
            return ProductionConfig
        elif mode == 'TESTING':
            from .testing import TestingConfig
            return TestingConfig
        else:
            from .development import DevelopmentConfig
            return DevelopmentConfig
    except ImportError:
        from .default import Config
        return Config
```

config/development.py

```
# coding: utf-8
import os


class DevelopmentConfig(object):
    """Base config class."""
    # Flask app config
    DEBUG = False
    TESTING = False
    SECRET_KEY = "sample_key"

    # Root path of project
    PROJECT_PATH = os.path.abspath(os.path.join(os.path.dirname(__file__), '..'))

    # Site domain
    SITE_TITLE = "twtf"
```

```
        SITE_DOMAIN = "http://localhost:8080"

        # MongoEngine config
        MONGODB_SETTINGS = {
            'db': 'the_way_to_flask',
            'host': '192.168.59.103',
            'port': 27017
        }

## 配置运行脚本

到此，我们的应用代码算是写完了，然后就是运行服务器了，还是使用 Flask-Scrip
t，所以我们需要配置 manage.py，内容为；

manage.py

#!/usr/bin/env python
# encoding: utf-8
from flask_script import Manager
from flask_script.commands import ShowUrls

from application import create_app

manager = Manager(create_app)
manager.add_option('-c', '--config', dest='mode', required=False
)

manager.add_command("showurls", ShowUrls())

if __name__ == "__main__":
    manager.run()
```

运行服务器

```
pyhton manage.py -c development runserver
```

当你看到以下语句的时候说明你的服务器运行成功了：

```
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

添加用户

因为现在数据库中是没有用户的，所以我们需要手动添加一个用户先，在管理后台可以添加：

```
http://localhost:5000/admin/user/
```

Admin Home User Role

List Create

name

password

email

role ---

Save | Save and Add Another | Save and Continue Editing | Cancel

Admin Home User Role

List Create With selected

		Name	Permission
There are no items in the table.			

测试功能：

登录：

```
POST /auth/login HTTP/1.1
Host: localhost:5000

{"username": "zhangsan",
 "password": "password"}
```

响应应该是：

```
{
  "email": "zhangsan@gmail.com",
  "name": "zhangsan",
  "role": "ADMIN"
}
```

使用 Gunicorn 和 Nginx 部署 Flask 项目

在实际的生产环境中，我们很少是直接使用命令：

```
python app.py
```

运行 Flask 应用提供服务的，正常都会集成 WSGI Web服务器提供服务，而在众多的 WSGI Web 服务器中，比较常用的主要有两种，分别是 Gunicorn 和 UWSGI，同时，我们也会使用 Nginx 作为反向代理进行部署应用。

本文因为需要安装 Nginx，所以文章内的命令和使用的系统相关，但是这样的命令不多，本文使用的 **Ubuntu 16.04**，因此包管理软件是 **apt**，如果使用的 RedHat 系列的话，那完全可以用 **yum** 代替。其他系列的系统可以查找相关文档寻找代替管理工具。

安装组件

```
sudo apt-get update
sudo apt-get install python-pip python-dev nginx

pip install gunicorn
pip install flask
```

这里前两句是更新一下软件源，并且保证我们的 pip 和 python 依赖库已经安装上了，同时，别忘了安装反向代理 Nginx。后面两句就是安装我们必备的 Gunicorn 和 Flask Python 库了。

下载代码

因为在我们的前文中已经写了一个代码了，所以这里就继续使用这段代码，使用方式是：

```
git clone git@github.com:luke0922/the-way-to-flask.git
cd the-way-to-flask/code
pip install -r requirements.txt
python manage.py runserver
```

此时，我们的服务器应该是已经运行起来了，但是，默认 Ubuntu 是开启了防火墙屏蔽所有端口访问的，所以我们可能需要打开防火墙端口，在 Ubuntu 16.04 中可以这样做：

```
sudo ufw allow 5000
```

现在，应该可以访问我们的应用了，在命令行上我们敲一下这个命令，访问以下 WEB 服务：

```
http://localhost:5000
```

一切正常的话，

创建 WSGI 切入点

```
vim wsgi.py
```

里面内容填：

```
from myproject import app

if __name__ == "__main__":
    app.run()
```

然后使用这个命令运行代码：

```
gunicorn --bind 0.0.0.0:5000 wsgi:app
```

依旧访问这个地址看看：

```
http://localhost:5000
```

常见 systemd Unit File

```
vim /etc/systemd/system/app.service
```

里面的内容写：

```
[Unit]
Description=Gunicorn instance to serve myproject
After=network.target

[Service]
User=www
Group=www
WorkingDirectory=/home/www/myproject
Environment="PATH=/home/www/myproject/myprojectenv/bin"
ExecStart=/home/www/myproject/myprojectenv/bin/gunicorn --worker
s 3 --bind unix:myproject.sock -m 007 wsgi:app

[Install]
WantedBy=multi-user.target
```

保存退出，然后尝试一下命令：

```
sudo systemctl start app
sudo systemctl enable app
```

配置 Nginx

配置Nginx

```
sudo nano /etc/nginx/sites-available/myproject
```

里面写：

```
server {
    listen 80;
    server_name server_domain_or_IP;

     location / {
        include proxy_params;
        proxy_pass http://unix:/home/sammy/myproject/myproject.s
ock;
    }
}
```

保存之后，用 nginx 自带工具验证一遍

```
nginx -t
```

如果ok的话然后让 nginx 重新加载配置

```
nginx -s reload
```

关闭服务器端口：

- sudo ufw delete allow 5000
- sudo ufw allow 'Nginx Full'

此时访问服务器试试：

```
http://192.168.59.103
```